

PROGRAMMING Expert

Global positioning systems may help you find your bearings, but you can still get lost when you try to connect a GPS to a PC. Mike James explains how to use a special protocol to download GPS data



MICROSOFT LAUNCHES VB WEBSITE

Microsoft has responded to protests about its lack of support for VB6 by launching a new website dedicated to classic VB. Last month we reported on an online petition demanding that Microsoft reconsider its VB strategy and it appears that Microsoft has listened. Its new VBRUN site at <http://msdn.microsoft.com/vbrun> has a Greatest Hits section containing old VB6 articles and a VB Fusion section that offers help on using VB6 and VB .NET together, as well as free training on using VB .NET. The site seems to be trying to persuade VB6 programmers that VB .NET is better. Microsoft still hasn't responded to complaints that legacy VB6, VBA and VBScript systems need continued support and not forced migration to VB .NET. Furthermore, there is still no help for VBA programmers who are left relatively unsupported because of the demise of VB6 and its associated ActiveX controls. If you want to sign the petition to keep VB6 available, visit <http://classicvb.org/petition>.

■ UNEMPLOYED? THEN WRITE OPEN SOURCE

A new Australian organisation is helping the unemployed to qualify for benefit by joining them up to open-source projects. In Australia you have to fulfil a mutual obligation requirement to receive unemployment benefit; this means you have to be involved in an activity that increases your chances of finding employment. Open-source projects are developed by programmers who are enthusiastic about an application and are willing to give their time for free. As a result we have operating systems such as Linux, applications such as OpenOffice and languages such as Mono, the open-source version of .NET. If this scheme of encouraging the unemployed to sign up with open-source projects works, it could catch on elsewhere. Visit www.communitycode.org for details.

In *Programming Expert*, *Shopper* February 2003 I explained how to create a program that read longitude and latitude readings from a global positioning system (GPS). Since then, GPS devices have become a lot cheaper. The project I looked at, which is reproduced on this month's cover disc, is a great way of finding your bearings. However, the National Marine Electronics Association (NMEA) protocol I used in the project is limited. Although this method of working with a GPS is widely compatible, it cannot access much of the information stored in smarter, standalone GPS units. If you want to connect a GPS to a PC and download positions on a continuous basis, NMEA is a reasonable choice. But if you want to take your GPS away with you, record a route and download it to a PC later, NMEA is no use.

The only way to get at data memorised by a GPS is by using whatever proprietary application program interface (API) is supported by the device. The biggest make of handheld GPS devices is Garmin and the Garmin API is available for download at www.garmin.com. If you read it, you might conclude that using it is far too difficult. The trick with complicated specifications is to interpret and implement them in a steady fashion, so each step builds on what you have already got working.

This month's project implements the Garmin protocol to download a track from a GPS and load it into an application. It is an excellent example of getting to grips with any API, so you should find it useful even if you have no interest in GPS.

PACKET MENTALITY

The Garmin protocol works by exchanging packets of data via a serial or Universal Serial Bus (USB) connection. The data structure is the same irrespective of the method of connection; only the details of opening and reading the data change. Here I'll use a serial connection because this is all my Etrex supports. The latest Garmin devices support both serial and USB. If your PC doesn't have a serial port, you can use the Garmin USB driver or a serial-to-USB converter.

Unfortunately Microsoft no longer supports Visual Basic 6 (VB6). This means it is difficult to get hold of a serial communications ActiveX control to use with Visual Basic for Applications (VBA), which is the obvious language to use if you want to import data into a spreadsheet, say. The MSComm

ActiveX control requires a VB6 licence to work. While there are other, similar ActiveX controls available, there is no obvious choice. A simpler option is to work in C#, which, in .NET 2.0, has a serial communications class. You can download C# Express from the Microsoft website and use it for free. However, you can program in C# using only Windows XP or later, though the programs you create run on Windows 98 or later. If you don't like C#, you should find the following code easy to translate into VB6 or any language of your choice.

Before we can work with these packets of data, we have to discover the structure of the packets and how they are exchanged. We also have to write a packet reader and packet writer. For details of the serial packet structure, see the table below. The USB connection uses a slightly different packet structure.

Every packet begins with a data link escape (DLE) character (ASCII 16) and has an ID code that defines what the rest of the packet is about. It also provides a byte size followed by a number of data bytes, then a checksum. The packet finishes with the DLE and the end of text (ETX) character. A complication is the use of 'DLE stuffing'. This replaces any occurrence of the DLE code in the body of the packet with two DLE codes. The second DLE isn't included in the size of the packet or the checksum and is stripped out by the receiver.

To make things easy, we need to find a simple part of the protocol where the PC sends the GPS a single packet and gets a single packet back in return. A quick look through the API documentation reveals that if the Packet ID (PID) sent is 254, the GPS sends back a single packet containing data that identifies its type. This is a good place to start.

DIFFERENT CLASS

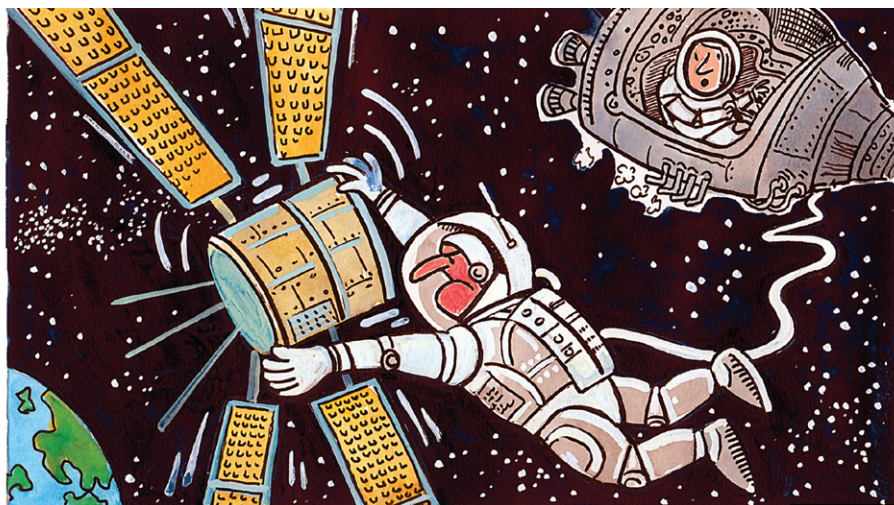
It makes sense to implement all the GPS functionality as a class. Start a new C# project and add to it a class called GPS. To use the serial port class that comes with .NET, we must add:

```
using System.IO.Ports;
```

First we need a constructor for the GPS object. For now, this just has to create and initialise the SerialPort class as com:

```
namespace WindowsApplication1
{
```

BYTE NO	BYTE DESCRIPTION	NOTES
0	Data link escape control character	ASCII DLE character (code 16)
1	Packet ID, identifies the type of packet	
2	Size of packet data (number of bytes)	
3	Packet data starts here and occupies 0 to 255 bytes	
n-3	A checksum (twos complement of the sum of bytes from byte 1 to byte n-4)	
n-2	Data link escape control character	ASCII DLE character (code 16)
n-1	End of text control character	ETX character (code 3)



'For the last time....it's your programming at fault and not the satellite !'

```
class GPS
{
private SerialPort com;

public GPS()
{
com = new SerialPort("COM1",
9600,
Parity.None, 8,
StopBits.Two);
com.Open();
}
}
```

The parameters are specified in the Garmin API document. Our next task is to write a packet-sending routine. Follow the specification given in the API. The routine hasn't been split into subroutines and only minimal error checking and handling are included. The routine starts as follows:

```
private void WritePacket(byte id,
byte[] data, byte size)
{
byte[] packet = new byte[1024];
int PacketSize = 0;
byte checksum = 0;
```

The parameters pass into the routine the packet's ID, the data array to be sent and the size of the array in bytes. The routine constructs a correct packet using the data byte array. Therefore the header is:

```
packet[0] = pid_dle;
packet[1] = id;
packet[2] = size;
PacketSize = 3;
```

The pid_dle constant is:

```
const byte pid_dle = 16; //ASCII DLE
code
```

We have to include the ID and the size in the checksum. This is computed using a simple sum truncated to a single byte:

```
checksum = (byte) (packet[1] +
packet[2]);
```

Next we transfer the data from the byte array to the packet, which is easy. However, having to perform DLE stuffing makes it more complicated:

```
if (size > 0)
{
for (int i = 0; i < size; i++)
{
checksum = (byte)(checksum +
data[i]);
packet[PacketSize] = data[i];
if (packet[PacketSize] ==
pid_dle)
{
PacketSize++;
packet[PacketSize] = pid_
dle;
};
PacketSize++;
};
}
```

The nested if statement performs the doubling up of DLE characters, but the additional DLE character isn't included in the checksum.

CHECK MATE

We bring the packet to a close by computing and storing the checksum and adding the DLE and ETX codes. The checksum is the twos complement of the total of all the bytes that make up the 'payload' of the packet. If you look up 'twos complement' you will discover that it can be computed by taking the bitwise Not and adding one:

```
packet[PacketSize] =
(byte)(~checksum + 1);
```

You can't move on yet because the checksum byte might be a DLE and so needs DLE stuffing:

```
if (packet[PacketSize] ==
pid_dle)
{
PacketSize++;
packet[PacketSize] = pid_dle;
};
PacketSize++;
```

Now we can just simply add the two final control codes:

```
packet[PacketSize] = pid_dle;
PacketSize++;

packet[PacketSize] = pid_etx;
PacketSize++;
```

Define the constant for the ETX code used like this:

```
const byte pid_etx = 3; //ASCII
ETX (End of text)
```

Now the packet is ready to be sent to the GPS. In our example, we can use the Write method of the SerialPort object. This accepts a byte array, an offset and how many bytes are sent:

```
com.Write(packet, 0, PacketSize);
```

For packet specification in the API, this is all we have to do. However, if you examine the sequence of packet exchange more carefully, you will discover that every packet sent is acknowledged by the GPS returning an ACK or a NAK packet. No more data can be sent until this happens. If the packet returned is a NAK packet, the data must be sent again. This also applies to every packet the GPS sends to the PC as we'll see.

It makes sense to tag this part of the logic on to the packet writer, but it shouldn't wait for an ACK/NAK packet if the packet it sent was itself an ACK/NAK packet:

```
if (id != 06 && id != 21)
{
packet = readpacket();
}
}
```

For simplicity, we are not even checking that the returned packet is an ACK/NAK packet, nor are we handling the retransmission needed for the NAK packet. For a full system, you should implement this. All you need to know is that an ACK packet has an ID of 06 and a NAK has an ID of 21. They also contain a single data byte that gives the ID code of the packet that failed, but as it's always the packet you just sent there's no point in examining this.

READER RIGHTS

The packet writer uses the packet reader and we now have to implement this routine. To provide the most general facility, this will return a byte array that contains the entire packet without any attempt to process it:

```
private byte[] readpacket()
{
int PacketSize = 0;
byte DLE, ETX, checksum;
byte[] packet = new byte[1024];
```

We form a loop reading in bytes until we detect a DLE code, which signals the start of a packet:

```
do
{
DLE = (byte)com.ReadByte();
}
while (DLE != pid_dle);
packet[0] = DLE;
```



Next we read the ID and the size of the data portion of the packet:

```
packet[1] = (byte)com.ReadByte();/  
/id  
packet[2] = (byte)com.ReadByte();/  
/size  
PacketSize = 3;
```

As we know how many bytes constitute the data portion of the packet, we can read this in. But remember to unstuff the DLE codes:

```
if (packet[2] > 0)  
{  
    for (int i = 0; i < packet[2];  
i++)  
    {  
        packet[PacketSize] =  
(byte)com.ReadByte();  
        if (packet[PacketSize] ==  
pid_dle)  
        {  
            packet[PacketSize] =  
(byte)com.ReadByte();  
        }  
        PacketSize++;  
    }  
}
```

Once the data is safely recovered, we can read the checksum and the terminating byte codes, remembering to unstuff the checksum:

```
checksum = (byte)com.ReadByte();  
if (checksum == pid_dle)  
{  
    checksum =  
(byte)com.ReadByte();  
}  
DLE = (byte)com.ReadByte();  
ETX = (byte)com.ReadByte();
```

Now we should compute the checksum. If it matches the transmitted checksum we should transmit an ACK packet, and if it doesn't we should transmit a NAK packet. Data errors are rare at 9600 baud with a standard three-foot cable, so we simply send an ACK packet:

```
if (packet[1] != 06 && packet[1]  
!= 21)  
{  
    byte[] data = new byte[1];  
    data[0] = packet[1];  
    WritePacket(6, data, 1);  
}  
return packet;
```

Again, we don't bother with ACK/NAK if the packet we just received was an ACK/NAK packet. However, we do construct the ACK packet correctly complete with the ID of the packet just received. Finally the packet is returned in its entirety.

IDENTIFYING YOUR GPS

Now we can send and receive a packet, we should see if they work by retrieving some information about the GPS on the other end of the connecting cable. The simplest to try is the A000 Product Data Protocol, which you can look up in the API

documentation. You have to send a packet with no data and an ID of 254. The GPS sends you back a packet with data containing two 16-bit integers – the product ID and the software version – and several null terminated strings that are a description of the device. To get this information, just send a packet with the appropriate ID:

```
public byte[] GetDescription()  
{  
    byte[] packet = new byte[1024];  
    Int16 pid, version;  
    WritePacket(pid_product_rqst,  
packet, 0);
```

The ID code is defined as:

```
const byte pid_product_rqst = 254;
```

We can rely on the packet sender to do everything else. All we do is read the GPS's reply:

```
packet = readpacket();
```

The first two data bytes make up the product identifier. To convert this to a 16-bit integer we can use the C# BitConverter object, which takes raw bytes and reassembles them into a range of data types. Converting the product identifier and software version is easy:

```
pid = BitConverter.ToInt16(packet,  
3);  
version = BitConverter.ToInt16  
(packet, 5);
```

To use the BitConverter, we must add:

```
using System.Windows.Forms;
```

Dealing with the strings is a bit trickier, because we don't know how many there are. A simple option is to read them all into a single C# string. In practice, most devices return a single string containing its name:

```
for (int i = 7; i <= packet[2];  
i++)  
{  
    description = description  
+ (char)packet[i];  
}  
return packet;
```

Add a button to the form and edit the code to read:

```
namespace WindowsApplication1  
{  
    public partial class Form1 : Form  
    {  
        private GPS gps;  
        public Form1()  
        {  
            InitializeComponent();  
            gps = new GPS();  
        }  
        private void button1_Click(  
object sender, EventArgs e)  
{  
    byte[] data;
```

```
data =  
gps.GetDescription();  
}  
}
```

If you run the program with the GPS connected and switched on, you should be able to verify that the data is retrieved from it. But as it stands, this isn't much use. It's better to transfer it to the clipboard so it can be pasted into another application. Add to the GPS constructor the line:

```
Clipboard.Clear();
```

At the end of the GetDescription routine, add:

```
AddClipLine(description);
```

THE RIGHT TRACK

You now have a system that can ask the GPS for something and retrieve it. It's time to tackle a more useful task. There are too many different commands defined within the API to cover them all, so here we'll deal with just one: retrieving the stored tracks. Look this up in the API before you read on. There are two possible versions, A300 and A301, depending on the GPS you are using. In fact, A300 is a subset of 301 so we can deal with both in the same way.

The new routine starts in the usual way by building a packet to give the transfer track command. But this time, the command is sent as a 16-bit integer in the first two data bytes:

```
public byte[] GetTrack()  
{  
    byte[] data = new byte[256];  
    byte[] packet = new byte[1024];  
    int numPackets=0;  
    data[0] = (byte)cmd_transfer_trk;  
    data[1] = (byte)(cmd_transfer_trk  
>> 8);  
    WritePacket(pid_command_data,  
data, 2);
```

The transfer track command must be defined as:

```
const Int16 cmd_transfer_trk = 6;  
//Get Track log
```

We can read and process the return packet, but this time there are multiple return packets. The first packet tells us how many packets to expect in the first two data bytes:

```
packet=readpacket();  
if (packet[1] == pid_records)  
{  
    numPackets = BitConverter.ToInt  
16(packet,3);  
}
```

If there are track packets to follow, each one contains a single point on the track in the same format. We have to read them all:

```
if (numPackets != 0)  
{  
    for (int i = 0; i < numPackets  
; i++)  
    {  
        packet = readpacket();
```




REVIEW BOOK

Beginning 3D Game Programming

RATING ★★★★★

PRICE £25.50

ISBN 0672326612

PAGES 448

Tom Miller's *Beginning 3D Game Programming* is an outstanding book. It contains errors, poor layout and misleading claims, but if you approach the book in the right way these problems don't matter. If you just want some code that you can type in and try out then you may be disappointed. But if you want a good start on how to use DirectX from C# or any managed code, it is ideal.

The book starts off by looking at complete games, but goes on to look at a set of illustrative examples. It uses the sample framework supplied with the most recent issue of the DirectX SDK. This framework is likely to be developed and should become the



standard way of programming DirectX as it reinvents itself as a pure class library.

This book claims to be for the beginner, but it isn't and its attempts to be accessible reduce its value. Still, it's an indispensable resource for anyone who wants to use DirectX from managed .NET code, even if writing 3D games isn't your main objective.

✓ Makes DirectX look easy from .NET languages

✗ Increasingly rushed as the book progresses

REVIEW BOOK

PHP for the World Wide Web: Visual QuickStart Guide

RATING ★★★★★

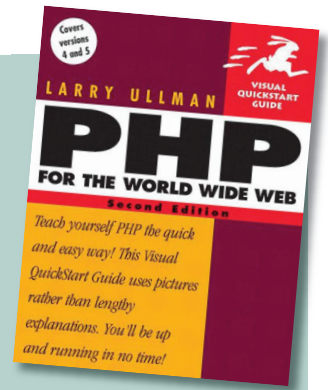
PRICE £17

ISBN 0321245652

PAGES 450

Larry Ullman's book on PHP is now in its second edition and it's good, but only if you are a complete beginner. It ignores the sophisticated aspects of PHP, including object orientation. "This topic is beyond the scope of this book" is a phrase that crops up frequently.

If you can live with that, this book should provide a useful introduction to PHP. It starts from the very basics of programming and progresses to the point where you should be able to do something useful. It covers the fundamentals of MySQL and using databases and files within web pages, but omits anything advanced.



If you can create a web page and want to learn how to create dynamic or data-based web pages, this is an excellent place to start. The explanations are thorough and the examples are usually helpful, though occasionally they could be simpler. The second edition clears up some of the errors in the first edition and extends coverage to PHP5 as well as PHP4.

✓ A well-paced introduction to using PHP4 and 5

✗ Doesn't cover anything even slightly advanced

The API documentation tells us that if there's more than one track, the start of each one is signalled by a special packet, a track header containing its name:

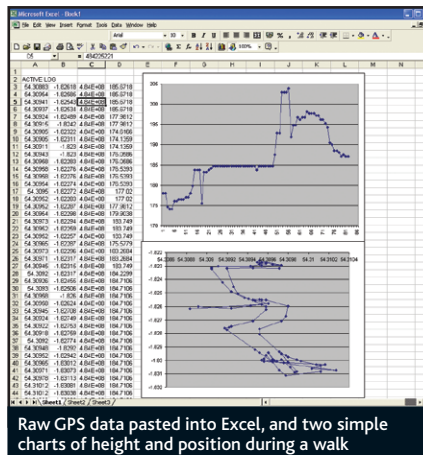
```
if (packet[1] == pid_trk_hdr)
{
    int j=5;
    String trackdescription = "";
    while(packet[j]!=0)
    {
        trackdescription =
        trackdescription+(char)packet[j];
        j++;
    }
    AddClipLine(trackdescription);
}
```

The AddClipLine routine simply adds the string to the clipboard as a new line:

```
private void AddClipLine(String text)
{
    text = text + "\n";
    Clipboard.SetDataObject(
    Clipboard.GetText()+ text,
    true);
}
```

If the packet is a track data packet then it contains the latitude and longitude as 32-bit integers:

```
if (packet[1] == pid_trk_data)
{
    Double latitude = BitConverter.
    ToInt32(packet, 3)
    / Math.Pow(2, 31) * 180;
    Double longitude = BitConverter.
    ToInt32(packet, 7)
    / Math.Pow(2, 31) * 180;
```



Both are measured in 'circles' and there are 2^{31} circles in 180 degrees, hence the conversion factors. The next four bytes denote the time in seconds:

```
Int32 time = BitConverter.ToInt32
(packet, 11);
```

Finally, the altitude is given in the final four bytes as a floating point value:

```
float alt = BitConverter.ToSingle
(packet, 15);
```

The track point can be added to the clipboard, complete with separating tabs so that it can be pasted into a spreadsheet or a word processor as a table:

```
AddClipLine(latitude.ToString() +
"\t"
+ longitude.ToString()+"\t"
+time.ToString()+"\t"
+alt.ToString());
```

```
}
}
}
return packet;
}
```

We also need to define the constants for the commands and ID bytes:

```
const byte pid_records = 27; //Records
packet
const byte pid_trk_hdr = 99; //Track
data header
const byte pid_trk_data = 34; //Track
data record
const byte pid_command_data = 10;
//Get data command
```

Add this line to the button's click routine:

```
data = gps.GetTrack();
```

If you connect a GPS that has some tracks stored in it and run the program, you should find that you can download them to the clipboard and paste the results to an Excel spreadsheet. If you just create plots of raw latitude against longitude, or height against time, you can immediately see where you have been. You can get much more from the clipboard data if you're prepared to make a few calculations, but then that's what a spreadsheet is for. **CS**

CONTACT

MIKE JAMES

Mike has written several books on computing and programming and has been a senior lecturer at Teesside University

mike@computershopper.co.uk